Cosmic Tracking in Real Time: Validating the Final Outer Tracker DAQ Chain

Dmytro Luchyn
Supervisor: Matteo Magherini

September 10, 2024

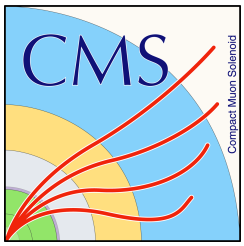
1 Introduction

The LHC is currently being prepared for the high luminosity upgrade. As part of of this upgrade, the CMS experiment will be introducing new silicon modules for the Outer Tracker. In order to verify the correct operation of the new sensors, tests with cosmic rays and test beams can be performed. Interpreting this data, however, can be challenging due to the non-standard file formats and difficulty in visualizing large amounts of said data.

This report will present a full-stack solution for working with 2S silicon sensor data. The stack involves a decoder for raw `.dat` files gathered from the readout FPGA boards, a tracker that performs alignment of the modules and builds likely particle tracks, as well as a web interface for interacting, decoding, tracking and visualizing the data in a user-friendly and efficient manner.

1.1 The Goal

The task of the application described in this report is to decode the raw binary files and re-package them into more user-friendly format for analysis, as well as reconstruct particle tracks based on the sensor data. The final application of the software will be in a cosmic ray detector used for the verification of modules and of the DAQ chain. Ideally it should be possible to use the software with files or even streams coming in in real time. The web interface should provide an easy interface for the backend software and allow the user to explore the generated track files.



2 Implementation Details

2.1 The Data Format

The file format generated by the acquisition computers is a direct consequence of the data format of the 2S modules themselves, which is in turn governed by their principle of operation. The name 2S stems from the fact that the module consists of two separate columns of silicon strips used to detect ionizing particles passing though them. Each module has 2 such layers, each side has 1016 strips, totaling 4064 strips per module. Two layers are required to correlate activation of the strips in order to filter out noisy activation and particles passing though the modules in orientations we are not interested in. When strips on both layers fire close in time and within a programmable window, then that is counted as a stub, and sent out for processing.

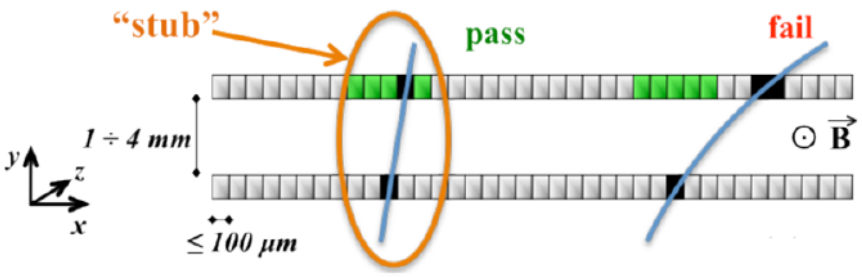


Figure 1: The stub generation process based on the two strip layers [1].

The process described above is performed by the CBC (CMS Binary Chip) ASICs. Each one has 254 readout channels connected to the strips on both layers of the detector. The strips are consciously monitored and if an event occurs that satisfies the above criteria, then the data about that event is forwarded onto the CIC (Concentrator Integrated Circuit) ASIC. This ASIC concentrates the data from the 8 CBCs per side of the detector, and then forwards the data to the LpGBT (Low power GigaBit Transceiver) to be sent down fiber optic lines to the acquisition system. Each 2S module has 16 CBCs, 2 CICs and 1 LpGBT.

After the data passes though the acquisition chain, it ends up packed into 1.2 GByte raw binary files. The file is built up in a hierarchical format: first, the stub sections are formed out of the strip address within the CBC, the CBC ID, the CIC ID, bend information, the link ID which corresponds to the module where the data came from and the bunch crossing (BX) offset. Then a collection of stubs is packed into a payload data structure. Each payload contains coarse bunch crossing information, called the BX Super ID, and fine bunch crossing within the LHC orbit time, simply called BX ID along with the number of stubs contained within this payload and some configuration and status bits. The exact ID of the event can be calculated as follows: `payload.header.bx_super_ID * 3564 + payload.header.bx_ID + stub.bx_offset` where 3564 is the LHC orbit time. This combined 64 bit value was called the bunch crossing ultra ID (BXUID) and will be used further in this report to refer to the time an event occurred. Each payload is ended by a 64 bit delimiter to identify the end of one and start of the next payload. These payloads are encoded into the last abstraction, the link header. Each link header contains a sequence number providing identification for the link header, as well as providing data on the length of the link header, the run number, and a type identifier. The link headers are distributed through the file, and are padded with zeroes until the next link header.

2.2 Decoding and Storage Strategy

The data format described above is easy to generate with data barreling in from an FPGA, but it is hard to work with for analysis purposes. Thus the first stage of the software was to re-package these files into a file format that would be denser than the 1.2GByte `.dat` binary files and simultaneously easier to use for analysis and track building. At first HDF5 was chosen to store the stub data in. However, it was quickly discovered that HDF5 increased the storage and processing overhead. This is due to HDF5 acting like a file system rather than a file format, each level of hierarchy costs 2.2 Kbyte in storage overhead [2]. The logical format to store the stub data, grouped by event had a large amount of hierarchy compared to the amount of data stored (each stub had only a few 8 bit values it needed to store). This resulted in the overhead being a large problem, since some .dat files can contain millions of events and with this overhead the resulting files will quickly outgrow the raw .dat files in size. Also, the C++ library that was used for HDF5 file interactions in the decoder ended up requiring disk access every time there was any write event, meaning there would be a large number of random disk IO during the writing process making the entire process very slow.

At this point, alternatives were considered and ultimately msgpack was chosen and tested. It offered numerous advantages in its simplicity and speed, which allowed the storage and retrial of large event files within seconds. msgpack is also a raw binary format with type hints meaning that while being very efficient by storing data in binary, it is still very accessible and can be opened without knowing the schema prior. The chosen msgpack library[2](link) also provides a very convenient way of turning a collection of structs into a schema by simply adding a macro to it. ¡example¿ After switching to msgpack, the format was also slightly flattened to improve performance further.

2.3 Debug View

After the decoding part was complete and there was a need to work on 3D data, a visualization was required in order to ensure that the data was correctly interpreted. Also, having the ability to look at the setup and each of its parts like stubs, and the track line over time during the line fitting and alignment operation would be invaluable to quickly identify and correct errors. Raylib was chosen for this purpose, it is an easy to use 2D and 3D rendering engine made primarily for video game development. The easy to use helper functions to draw common geometric shapes as well as the very simple structure of the library meant that it was perfect for this purpose. By using the known dimensions of the modules as well as the locations described in json file it is trivial to compute the vertices of each module´s layers and place it in 3D space of the scheme.

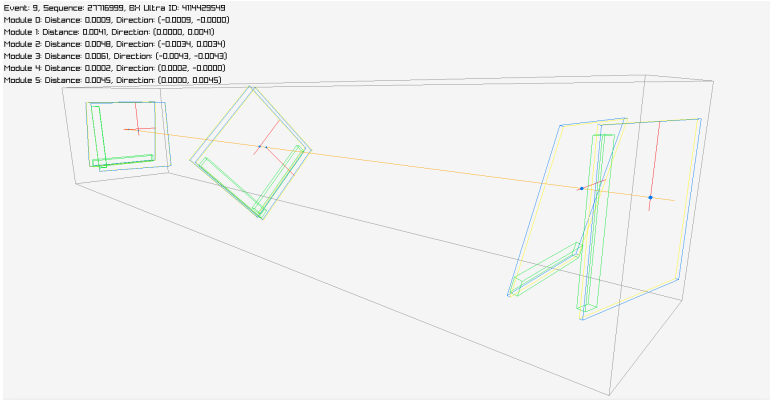


Figure 2: The debug view of the tracker software.

2.4 Tracking and Alignment

The primary objective of the software project is the ability to reconstruct particle tracks. The tracker is implemented as a least square minimizer using ceres [3], the cost function for which is defined as the distance between the stub strip and the point of the candidate line intersecting with the module plane.

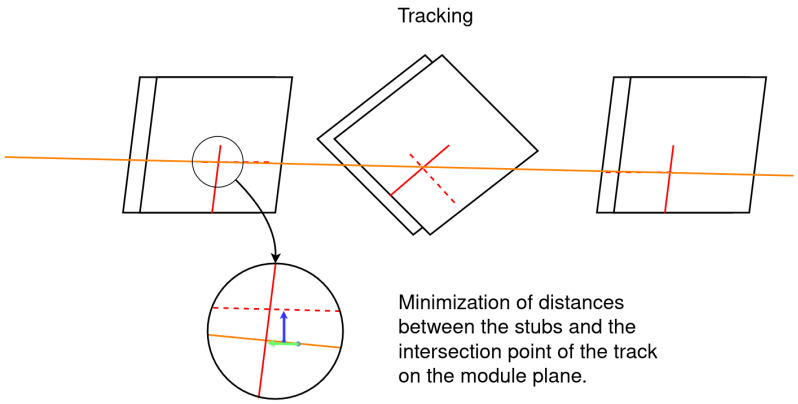


Figure 3: Diagram explaining the tracking process.

The tracker cannot reliably operate without the modules being aligned. This is performed before tracking the data for the entire file. Before alignment, all the events are filtered to only include events where there is exactly one stub per module. This excludes a lot of noise, but also some potential double track events. This was done for simplicity, but can be addressed in the future. Alignment is performed by iteratively selecting a random module to align, next a set of tracks is built without using the module for line building. After the tracks are built, the distance between the strip on the excluded module and the track is calculated and the module is moved in the direction. This is performed many times until all the modules are aligned.

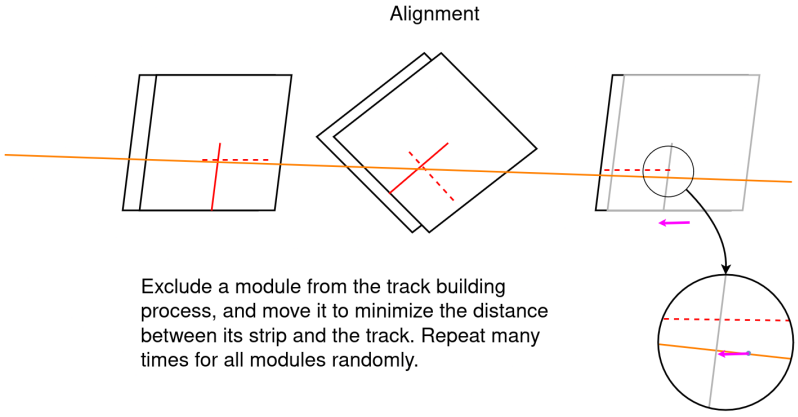


Figure 4: Diagram explaining the alignment process.

After a successful alignment, when using all the modules again for track building, the tracks are offset usually by less than the strip width of 90um. When the alignment stage has been complete, the tracker is used on the entire file to build tracks.

2.5 Web Interface

All of the low level processing as described above is performed in the C++ backend. This backend is compiled into a single binary executable that can be used with command line arguments. The backend is then wrapped in a python script that exposes an API with the help of the Flask [4] library. The API is then used by the frontend to interact with the backend. The frontend is a simple web interface that allows the user to interact with the files, request processing, and visualize the results.

The front end is implemented with a simple HTML and CSS layout, JavaScript for the interactivity, Three.js [5] for the 3D rendering and msgpack.js [6] for the track file parsing. The msgpack.js library was capable of parsing track files containing millions of events in a few seconds, which is more than enough for the end goal of looking at cosmic ray data, which is usually less frequent and thus the files will be less dense. After a track file is loaded, a list is created containing all of the events and their IDs. This list is presented to the user. The user can click the event of interest, which will display the event information and plot the stubs and the track in the 3D view. It is also possible to switch between events quickly using the left and right arrow keys, for quick comparison of events.

Figure 5: Web UI of the tracker software.

2.6 Containerization

In order to simplify the deployment of the software, a Docker container was created. The container contains all the necessary dependencies and the software itself. When the container is built, the C++ software is compiled and made accessible to the Python API script. Nginx [7] is also configured to serve the frontend, file downloads and the API. Everything is accessible through the 8080 port on the host machine, and can be exposed to an intranet for remote access within the organization.

Containerization also allows a single machine to run multiple instances of the software which can be useful for allowing each user or group of users to have their own isolated instance of the software.

It should be noted that the container is not meant to be exposed to the wider internet since it lacks any authentication or security measures. It is meant to be used within a controlled environment.

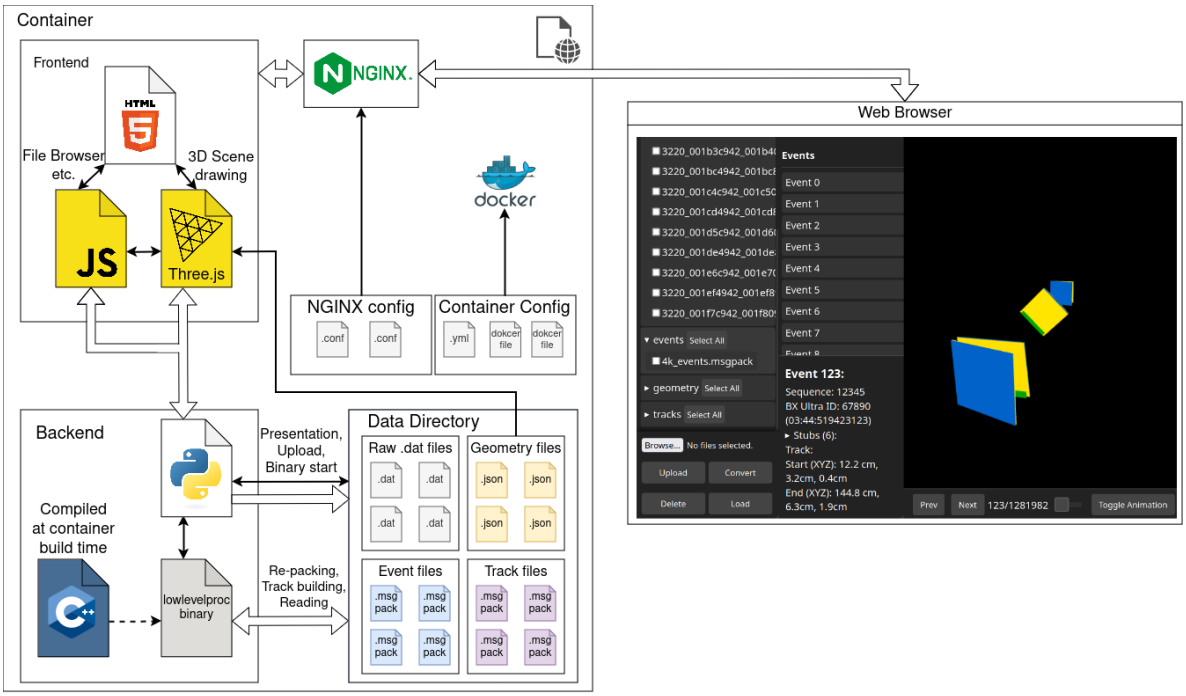


Figure 6: Block diagram of the software container structure.

3 Results

The final version of the software was tested for performance and usability. The software was able to process a 1.2 GByte file containing 3 million events in under 10 seconds on a AMD Ryzen 7 3700U. The alignment and tracking steps are computationally expensive and thus were multithreaded to bring the processing time down as much as possible. On the same CPU, it took 1 hour to reconstruct 2 million tracks from the same file. The final track file was 180 MBytes in size, and can be loaded into the web interface directly in under 10 seconds.

Since the test data was from a particle beam experiment, and thus had many more tracks than a cosmic ray experiment would have, the software is expected to perform even better in a real cosmic ray experiment.

4 Conclusion

In conclusion, the software was able to successfully perform the initially set goals of going from raw binary data to a fully interactive track visualization. The software is able to process large files in a reasonable amount of time and the web interface abstracts away the complexity of the backend software. The software is ready for deployment in the cosmic ray detector and will be used for the verification of the new modules and the DAQ chain.

References

[1] The Phase-2 Upgrade of the CMS Tracker. Tech. rep. Geneva: CERN, 2017. DOI: 10.17181/CERN.QZ28.FLHW. Available: https://cds.cern.ch/record/2272264

[2] StackOverflow Answer. "HDF5 Storage Overhead" Available: https://stackoverflow.com/a/15286009

[3] Ceres Solver. Available: http://ceres-solver.org/

[4] Flask Library. Available: https://flask.palletsprojects.com/

[5] Three.js Library. Available: https://threejs.org/

[6] msgpack.js Library. Available: https://github.com/ygoe/msgpack.js

[7] Nginx. Available: https://nginx.org/en/